МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЁТ

по лабораторной работе №5

по дисциплине «Алгоритмы и структуры данных»

Тема: Код Хаффмана

Студент гр. 8304		Воропаев А.О.
Преподаватель		Фирсов М.А.

Санкт-Петербург

2019

Задание.

Вариант №4

Статическое декодирование кода Хаффмана.

Цель работы.

Научиться программировать бинарные деревья.

Описание алгоритма.

После считывания функция make_dict создаёт словарь, где в качестве ключа выступает сам символ, а в качестве значения – его код. Затем мы формируем бинарное дерево путем прохода по каждому символу в словаре и добавлению его в дерево. Затем с помощью функции decode мы декодируем зашифрованную строку путем прохода по дереву.

Описание функций программы:

```
1. void read_file(std::vector<std::string>& file_data, std::ifstream& input)
```

Функция предназначена считывания данных из файла.

file_data – вектор, куда будут записаны данные из файла

input – файловый поток ввода

```
2. bool make_binary_tree(std::string code, char c, std::shared_ptr<Node>&
head)
```

Функция, добавляющая элемент в бинарное дерево по данному ей коду элемента.

code – код элемента, по которому будет произведена его запись в дерево

head – ссылка на корень бинарного дерева.

```
3. bool make_dict(std::string& input_codes, std::map<char, std::string>&
dict)
```

Функция, создающая словарь по входной строке пар типа символ-код

input_codes – входная строка пар символ-код

dict – ссылка на словарь

```
4. bool decode(std::string const& code, std::shared_ptr<Node>& head,
std::string& result)
```

Функция, декодирующая закодированное сообщение

code – строка, содержащая закодированное сообщение

result – строка, в которую будет записан результат декодирования

Выводы.

В ходе работы был получен опыт работы с кодом Хаффмана.

Протокол

Тестрование:

Входные данные	Выходные данные
000100100011010101100111100010011010010110111 ((m 0001)(a 0010)(g 0011)(i 0101)(c 0110)(h 0111)(e 1000)(s 1001)(k 1010)(y 1011))	magicheskiy
0000 ((A 0)(A 1))	Collision!
000011 ((U 1)(B 01)	Incorrect code
Efgwrh4rydtrjdry ((C 1000))	Incorrect symbol in the coded message
10101010101010101001)(	Error/make_dict
001 ((A 001))	A

Исходный код

Main.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <map>
#include <regex>

struct Node
{
    Node() = default;

    char c = '0';
    std::shared_ptr<Node> left;
    std::shared_ptr<Node> right;
};
```

```
void read_file(std::vector<std::string>& file_data, std::ifstream& input)
{
    std::string current_file_string;
    while (std::getline(input, current_file_string))
    {
        if (current_file_string.back() == '\r')
            current_file_string.erase(current_file_string.end() - 1);
        file_data.push_back(current_file_string);
    }
}

bool make_binary_tree(std::string code, char c, std::shared_ptr<Node>& head){

        std::shared_ptr<Node> processing_node(head);

        for (char element : code)
        {
            if (processing_node->c != '0'){
                std::cout << "Bad prefix" << std::endl;
                return false;
            }
            if (element == '0')
            {
                if (processing_node->left == nullptr)
                {
                    auto new_node = std::make_shared<Node>();
                    processing_node->left = new_node;
                }

                processing_node = processing_node->left;
            }
            else if(element == '1')
            {
                if (processing_node->right == nullptr)
                {
                    auto new_node = std::make_shared<Node>();
                    processing_node->right = new_node;
                }
                processing_node = processing_node->right;
            }
        }
        if (processing_node->c != '0'){
            std::cout << "Bad prefix" << std::endl;
            return false;
        }
        processing_node->c = c;

        return true;
}

bool make_dict(std::string& input_codes, std::map<char, std::string>& dict){

    std::regex pattern("\\(\\w [01]+\\)");
    std::smatch match;

    while (std::regex_search(input_codes, match, pattern) != 0)
    {
        std::string correct_pair(match.str());
        std::string characterCode(correct_pair.begin() + correct_pair.find("
") + 1, correct_pair.begin() + correct_pair.find(")"));
        char character = correct_pair[1];

        if (dict.find(character) != dict.end()){
```

```
            std::cout << "Collision!\n_____________________________________\n";
            return false;
        }

        dict[character] = characterCode;

        input_codes.erase(match.position() + input_codes.begin(),
match.length() + match.position() + input_codes.begin());
    }
    if(dict.empty()){
        std::cout << "Er-
ror/make_dict\n_____________________________________\n";
        return false;
    }
    return true;
}

bool decode(std::string const& code, std::shared_ptr<Node>& head,
std::string& result){

    std::string current_checked_code;
    std::shared_ptr<Node> processing_node(head);

    for(char c : code) {

        if(c == '1') {
            if(processing_node->right == nullptr) {
                return false;
            }
            if(processing_node->right->c != '0') {
                result += processing_node->right->c;
                processing_node = head;
            }
            else
                processing_node = processing_node->right;
        }

        else{
            if(processing_node->left == nullptr) {
                return false;
            }
            if(processing_node->left->c != '0') {
                result += processing_node->left->c;
                processing_node = head;
            }
            else
                processing_node = processing_node->left;
        }
    }
    return true;
}

int main(int argc, char* argv[]) {

    if (argc >= 2)
    {
        std::ifstream input(argv[1]);
        if (!input.is_open())
        {
            std::cout << "Incorrect input file\n";
            return 1;
        }

        std::vector<std::string> file_data;
```

```
        read_file(file_data, input);

        for (int i = 0; i != file_data.size(); i += 2)
        {

            std::string code_string = file_data[i];
            std::string input_pairs = file_data[i+1];

            std::cout << "Test #" << (i / 2) + 1 <<
            "\nCoded message: " << code_string <<
            "\nInput symbol-code pairs: " << input_pairs << std::endl;

            std::string check_str = "10";
            if(code_string.find_first_not_of(check_str) !=
std::string::npos){
                std::cout << "Incorrect symbol in the coded mes-
sage\n_____________________________________\n";
                continue;
            }

            std::map<char, std::string> dict;

            if (!make_dict(input_pairs, dict)) {
                continue;
            }

            auto head = std::make_shared<Node>();

            for (auto& pair : dict)
            {
                char character = pair.first;
                std::string code = pair.second;
                if (!make_binary_tree(code, character, head))
                {
                    std::cout << "Error/make_bi-
nary_tree\n_____________________________________\n";
                    continue;
                }
            }

            std::string result;
            if (!decode(code_string, head, result))
            {
                std::cout << "Error/decode(incorrect
code)\n_____________________________________\n";
                continue;
            }
            dict.clear();
            std::cout << "Result: " <<  result <<
"\n_____________________________________" << std::endl;
        }
    }
    return 0;
}
```